Application Note 201

Building and Debugging ARM Linux Using ARM Embedded Linux, ARM RealView Development Suite 3.1 and RealView ICE 3.2

ARM

Application Note 201
Building and Debugging Embedded Linux

Copyright © 2008 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

<div style="text-align:right">Change history</div>

Date	Issue	Change
June 2008	A	First release

Proprietary notice

Words and logos marked with ® or © are registered trademarks or trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Contents

Copyright © 2008 ARM Limited. All rights reserved.

1   Introduction

1.1   Support status

Please note that ARM does not provide support on the use of GNU tools or Linux. The information provided here is given for your reference only. We can provide limited support for the instructions in this document to customers with a valid RealView tools support contract with us. However, we suggest that in the first instance you discuss your issue in one of the various public forums, such as the comp.sys.arm newsgroup and the ARM website forums.

Alternatively you may prefer to contact CodeSourcery, who can provide paid support and assistance on the GNU tools or accept defect reports. Further information is available from CodeSourcery's GNU toolchain page at:
http://www.codesourcery.com/gnu_toolchains/arm/.
CodeSourcery also provide a mailing list for queries on the ARM GNU toolchain. Details of how to subscribe to this list can be found on the CodeSourcery website.

1.2   Scope of this document

The document describes the process of building an ARM Linux kernel and bringing it up on an ARM hardware target. It also describes the process of debugging applications using Eclipse, the GNU Project Debugger (GDB) and gdbserver, and the process of debugging the kernel and statically linked drivers using RealView Debugger.

This document assumes a specific target system (see **Section 1.3**). It should be possible to adapt the instructions in this document to work for different target systems and the instructions are written generically where possible.

This document does not cover the use of RealView Compilation Tools for building applications or libraries for an ARM Linux system: this is covered by **Application Note 178: Building Linux Applications Using RVDS 3.1 and the GNU Tools and Libraries** (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0178a/index.html).

Expected use

This application note supports the following use cases:

- Porting a Linux kernel to a new target

- Developing Linux applications on an existing ARM Linux target (in conjunction with Application Note 178)

1.3   Requirements

This document assumes that you are familiar with RealView Development Suite, the GNU tools and Linux.

Target requirements

This document assumes a target system composed of a RealView Versatile PB926 board (hereafter referred to as a PB926) and ARM Embedded Linux 2.5.0 ('Freetown'), which uses the 2.6.24 Linux kernel with the 2.6.24-arm2 patch applied. This is just in order to make the scope of what is described here manageable and to avoid repeated digression into target-specific differences. The instructions in this document can be adapted to work for different target systems.

ARM Embedded Linux can be found on the ARM website at: http://www.arm.com/products/os/linux_download.html.

Please note that ARM Embedded Linux is an example ARM Linux distribution only and is not a supported product.

Development environment requirements

All information in this document relates to the use of RealView Development Suite 3.1, RealView ICE 3.2 and the 2007q1-21 release of CodeSourcery's GNU ARM toolchain. In most cases, the host environment may be assumed to be any version of Linux or Windows supported by these products. The only exception to this is **Section 2: Building the ARM Embedded Linux Kernel**, which assumes a Linux host.

Later versions of the CodeSourcery toolchain may work, but this application note is validated only against the 2007q1-21 release.

All sections use Linux syntax to refer to paths and environment variables on the host machine (path components are separated with /, not \, and environment variables are presented as $<name>, for example $ARMROOT refers to the ARMROOT environment variable).

All screenshots are taken from a Windows XP host.

2 Building the ARM Embedded Linux Kernel

2.1 Requirements

This section assumes that you are building the kernel on a Linux host machine. Building the Linux kernel on a Windows host is beyond the scope of this document.

To build the ARM Embedded Linux kernel from source you will require the main-line kernel source for the 2.6.24 kernel, the ARM-specific patch for the main-line kernel source and the ARM-specific kernel configuration files. The kernel source can be obtained from one of the mirrors listed on http://www.kernel.org/mirrors/, while the other required files can be obtained from http://www.arm.com/products/os/linux_download.html.

To perform the build you will need the 2007q1-21 release of the CodeSourcery toolchain for the ARM GNU/Linux target platform. This is available from http://www.codesourcery.com. At time of writing it can be found at http://www.codesourcery.com/gnu_toolchains/arm/releases/2007q1-21. The following instructions assume that you have already installed the CodeSourcery toolchain. These instructions are likely to work with later releases of the CodeSourcery toolchain, but have only been validated against the 2007q1-21 release.

To build the U-Boot kernel image, you will need the U-Boot `mkimage` utility, which is linked to from http://www.arm.com/products/os/linux_download.html: it should be downloaded as a `blob`. At time of writing it can be downloaded directly from http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=u-boot/tools/mkimage.

Files required for all targets are listed in **Table 2-1**, while the required files for a PB926 target are listed in **Table 2-2**. Equivalent files for a different target can be downloaded in place of the files listed in **Table 2-2**.

Note You should ensure that you get the source for the original 2.6.24 kernel release and not any of the later patch releases (for example 2.6.24.1) as the ARM patch may produce incorrect code if applied to later releases. The 2.6.24 release is the *last* 2.6.24 kernel listed on kernel.org.

Note The `mkimage` file will download as `u-boot-tools-mkimage`. You will need to change the name of the file to `mkimage` (`mv u-boot-tools-mkimage mkimage`). You may also need to set the executable flag on the file (`chmod u+x mkimage`).

Note If you use Microsoft Internet Explorer to access the ARM Embedded Linux files, files may download with the name `git`. The downloaded files can be renamed according to **Tables 2-1** and **2-2**, either after download or by right-clicking and using the `Save as...` option.

Table 2-1 Required Files for All Targets

Address at time of writing	Name of downloaded file	Description
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.tar.bz2	`linux-2.6.24.tar.bz2`	Main-line kernel source.
http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=kernel/src/patch-2.6.24-arm2.gz;hb=2008q1	`kernel-src-patch-2.6.24-arm2.gz`	Patch to apply to main-line kernel source.
http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=u-boot/tools/mkimage	`u-boot-tools-mkimage`	Tool to prefix kernel image with a U-Boot header. Must be renamed to `mkimage` after download.
http://www.codesourcery.com/gnu_toolchains/arm/releases/2007q1-21	`arm-2007q1-21-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2 (Linux)`	CodeSourcery ARM GNU/Linux toolchain.
	`arm-2007q1-21-arm-none-linux-gnueabi.exe (Windows)`	

Table 2-2 Required Files for PB926

Address at time of writing	Name of downloaded file	Description
http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=kernel/config/2.6.24-arm2/config-2.6.24-arm2-versatile;hb=2008q1	`kernel-config-2.6.24-arm2-config-2.6.24-arm2-versatile`	Kernel configuration file for ARM Versatile boards.

2.2   Building the Kernel

2.2.1   Prepare the Working Directory

Ensure that you have sufficient space available to build the kernel in. The kernel source and object files alone are likely to require a minimum of 600 MB of space, and this minimum may well increase if you add extra drivers or other patches. Download the required kernel source archive and patch, plus an appropriate configuration file for your target board and a copy of `mkimage`. These files should be placed in a working directory on the host where you intend to build the kernel. This directory will be referred to as `<workdir>` from now on.

Extract the kernel source archive (`tar xvjf linux-2.6.24.tar.bz2`) and the ARM patch for the kernel source (`gunzip kernel-src-patch-2.6.24-arm2.gz`). After extraction your `<workdir>` should contain a full kernel source tree along with several other files. These files and directories are listed in **Table 2-2**.

Table 2-2 Key contents of <workdir>

File/Directory	Description
`linux-2.6.24`	Directory containing kernel source
`kernel-src-patch-2.6.24-arm2`	Patch to be applied to the kernel source in order to build a kernel suitable for ARM targets
`kernel-config-2.6.24-arm2-config-2.6.24-arm2-versatile`	Configuration file for target board (file name will be different if the target is not a PB926)
`mkimage`	Utility to create U-Boot images from kernel images

2.2.2   Set up the Build Environment

Put `<workdir>` on your path, so that the `mkimage` utility can be found.

If you do not have the CodeSourcery GNU tools on your path, set the environment variable `CROSS_COMPILE` to the location of your CodeSourcery `bin` directory, followed by `arm-none-linux-gnueabi-` for example

```
export CROSS_COMPILE=~/codesourcery/bin/arm-none-linux-gnueabi-
```

If the CodeSourcery GNU tools are on your path then you do not need to set `CROSS_COMPILE`.

2.2.3   Prepare and Build the Kernel

Enter the `<workdir>/linux-2.6.24` directory and apply the ARM patch (`patch -p1 < ../kernel-src-patch-2.6.24-arm2`). Then copy the configuration file into place (`cp ../kernel-config-2.6.24-arm2-config-2.6.24-arm2-versatile .config`).

Run the command `make oldconfig` to import the configuration settings from the new configuration file. If you want to modify the kernel configuration you can do this in the normal way (for example `make menuconfig` or `make xconfig`).

Finally, run `make uImage` to have the build process generate a kernel image with a U-Boot header. When the build completes, the directory `<workdir>/linux-2.6.24/arch/arm/boot/uImage` will contain the kernel images listed in **Table 2-3**.

Table 2-3 Linux kernel images in arch/arm/boot

File	Description
Image	Raw binary kernel image
zImage	Self-extracting compressed binary kernel image
uImage	U-Boot kernel image, composed of zImage plus a U-Boot header

3 Installing & Running ARM Embedded Linux on the Target System

3.1 Requirements

This section assumes that you have either collected the Linux system binaries from the ARM Linux web site (http://www.arm.com/products/os/linux_download.html), or else built them yourself as described in **Section 2**. If using images that you have built yourself, or pre-built images for a target other than the PB926, then you will need to substitute file names appropriately throughout this section. The required files are listed in **Table 3-1**. Note that you should use only one of the file system images, and that the Thumb-2 file system image requires a target that can execute Thumb-2 instructions.

This method of installing ARM Embedded Linux relies on the use of BootMonitor to pass control to the U-Boot bootloader, and on the use of ARM's Eclipse Flash Programmer to program images to flash. The ARM Eclipse Flash Programmer is available in the latest Eclipse update for RealView Development Suite 3.1 and later. If you do not have the flash programmer then ensure that you have updated your Eclipse installation to the latest version of the RealView Development Suite Plug-ins. See **Installing software updates for the Eclipse Plug-ins for RVDS** in the **RealView Development Suite Eclipse Plug-in User Guide** for details of how to do this.

There are many alternative approaches that can be taken to install and boot Linux, including booting from a network, using an NFS root file system, using U-Boot without BootMonitor to boot the system, etc. All alternative approaches are considered to be outside the scope of this document.

Note   If you use Microsoft Internet Explorer to access the ARM Embedded Linux files, files may download with the name `git`. The downloaded files can be renamed according to **Tables 2-1** and **2-2**, either after download or by right-clicking and using the `Save as...` option.

Table 3-1 Linux system files

Address at time of writing	Name of downloaded file	Description
http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=kernel/bin/2.6.24-arm2/uImage-2.6.24-arm2-versatile;hb=2008q1	`kernel-bin-2.6.24-arm2-uImage-2.6.24-arm2-versatile`	Pre-built kernel image for Versatile boards with an ARM926EJ-S core
http://www.linux-arm.org/git?p=ael.git;a=blob_plain;f=u-boot/bin/u-boot_versatilepb.axf;hb=2008q1	`u-boot-bin-u-boot_versatilepb.axf`	U-Boot image for Versatile PB boards
http://www.arm.com/linux/armbase_2.5.cramfs	`armbase_2.5.cramfs`	Base AEL file system
http://www.arm.com/linux/armfull_2.5.cramfs	`armfull_2.5.cramfs`	Full AEL file system
http://www.arm.com/linux/t2_base_2.5.cramfs	`t2_base_2.5.cramfs`	Base AEL file system compiled as Thumb-2

3.2 Installing and Running the System

3.2.1 Check BootMonitor

The target board must have a working Boot Monitor in Flash. To verify this:

1 Connect UART0 on the development board to a serial port on your computer using the NULL modem cable supplied with your development board.

2 Configure a terminal emulator (such as HyperTerminal, TeraTerm or minicom) to connect to the serial port. The settings should be: baud rate 38400, 8 data bits, no parity, 1 stop bit and no flow control.

3 Turn on the power to your board. You should see a startup message similar to the following in your terminal emulator:

```
ARM PB926EJ-S Boot Monitor
Version:   V4.0.6
Build Date: Apr 17 2007
Endian:    Little
```

If your boot monitor does not work, follow the instructions in the user guide for your development board (**Getting Started** section) to program a Boot Monitor into your board. This guide includes the DIP switch settings to make Boot Monitor run.

3.2.2 Determine Available Space in Flash

Once you have a working Boot Monitor, determine the first available free space in flash.

1 Type the following at the Boot Monitor prompt:

```
> Flash
Flash> list areas
```

This should display something like:

```
Base        Area Size Blocks Block Size
----        --------- ------ ----------
0x34000000      256K      4        64K
0x34040000    65280K    255       256K
0x30000000      256K      4        64K
0x30040000    65280K    255       256K
```

This tells you the total amount of flash available and the size of each block of flash.

2 Determine the images already in flash. To do this type:

```
Flash> list images
```

This should display something like:

```
Flash Area Base 0x34000000

Address     Name
-------     ----
0x34000000  Boot_Monitor
0x34030000  SYSTEM.DAT
```

3 Find out how much space is taken up by the last image in Flash. In this case, you would type:

```
Flash> display image SYSTEM.DAT
```

This should display something like:

```
Name: SYSTEM.DAT
Flash Address: 0x34030000
Load Address : 0x34030000
Entry Point  : 0x34030000
Size         : 56 Bytes
Blocks Used  : 1
```

4   You now have sufficient information to calculate the first free space in flash. In this case, SYSTEM.DAT exists in flash at address 0x34030000. It is using 1 block of flash and the block size is 64K, which is 0x10000 bytes. Therefore the first free block begins at address 0x34040000 (0x34030000 + 0x10000). Note that in this example, this is beginning of a new area of flash with a larger block size (256K, or 0x40000 bytes).

### 3.2.3   Program the System Images into Flash

There are many ways of programming images to Flash, but as this application note is focused on the use of Eclipse we will describe the use of ARM's Eclipse Flash Programmer. If you are using an ARM Versatile Board then you could use BootMonitor instead. Refer to the appropriate section of your board's user guide for more information (**Section 2.6: Using the PB926EJ-S Boot Monitor and platform library** in the **RealView Platform Baseboard for ARM926EJ-S User Guide** in the case of the PB926).

1   Ensure that you have downloaded all of the system binaries listed in section 3.1.

2   Start Eclipse. Ensure that you are in the C/C++ Perspective. If you are not, you can change to it by selecting Open Perspective from the Window menu.
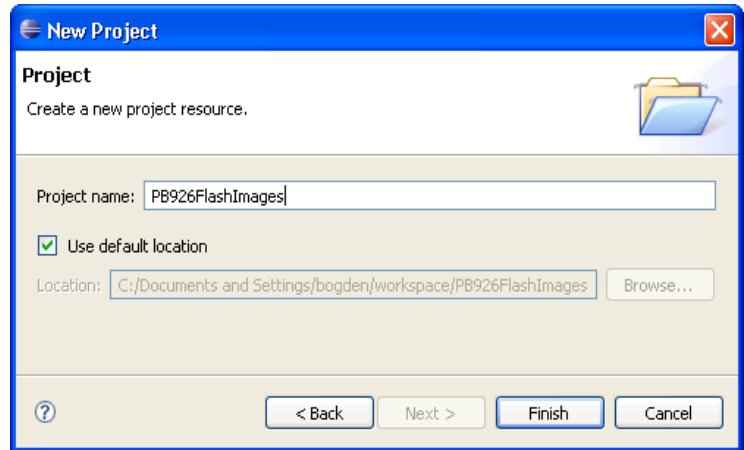

Shows C/C++ Perspective

3   You will need to create a project to store your system binaries for upload to your target. To do this select New->Project... from the File menu to bring up the New Project dialog. Expand the General node in the tree view and select Project. Click Next.

Application Note 201
ARM DAI 0201A
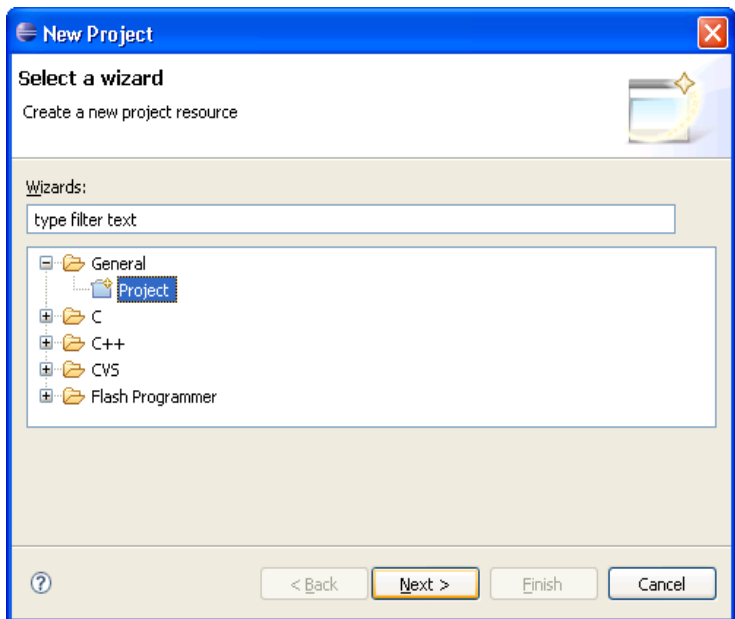
4   On the new screen, give the project a name and click `Finish`.

5   Right-click on your new project in the `C/C++ Projects View` and select `New->Folder` from the context menu to open the `New Folder` dialog.

       Application Note 201
   ARM DAI 0201A
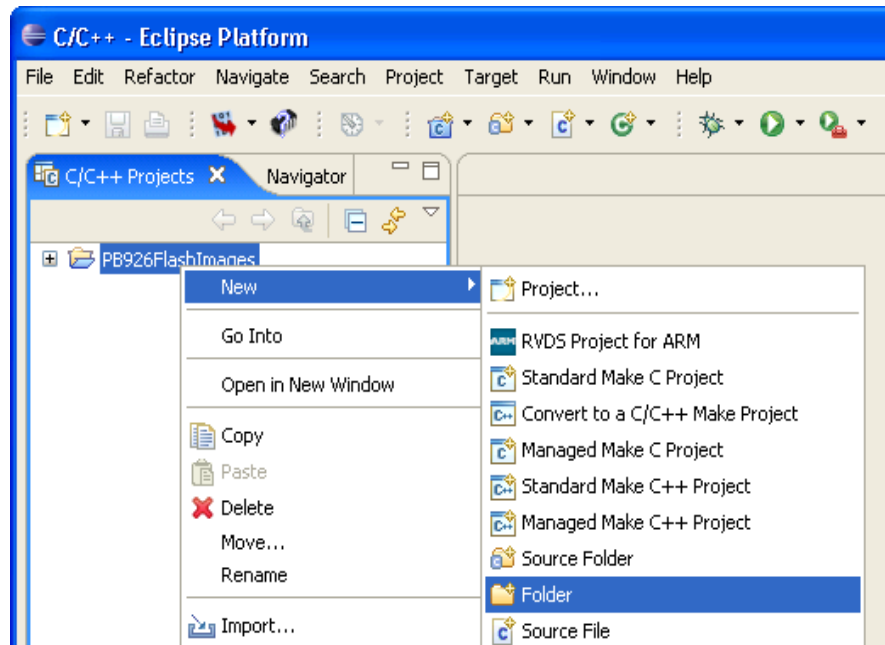
6   Give the folder a name (for example `System Binaries`) and press the
`Advanced >>` button. Tick `Link to folder in the file system` and
press the `Browse...` button. Browse to the location where you have stored the
system binaries in the `Browse For Folder` dialog. If you have built a kernel
from source then the directory should also contain the kernel's build tree. Press
`OK` to return to the `New Folder` dialog and then press `Finish`.
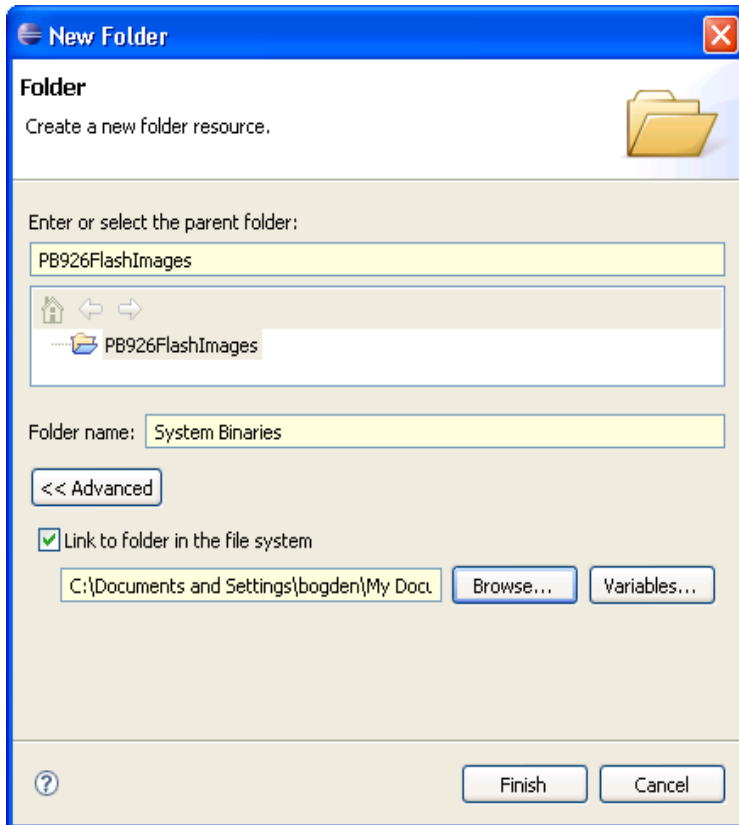
7   Your project should now contain the folder that you have created, and this folder
should contain the system binaries that you have downloaded.

Program U-Boot Image

1 Navigate to the U-Boot image in your project (for example, `System Binaries/u-boot-bin-u-boot_versatilepb.axf`). Right-click on this image and select
`Send To->Send To...` from the context menu to open the `Send To` dialog.
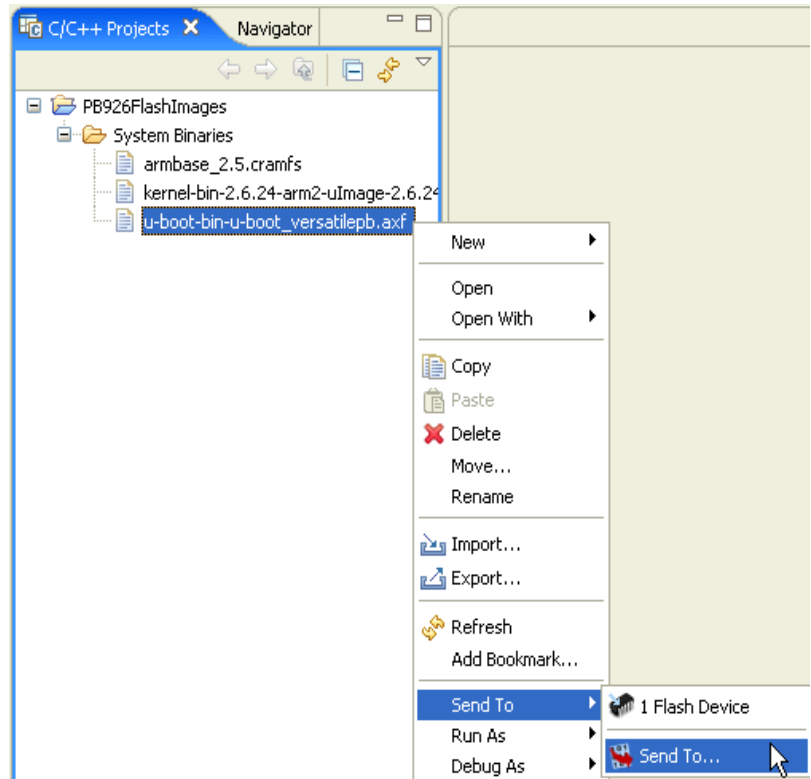
2 In the `Send To` dialog, select `ARM Flash Programmer` in the tree view and press the `New` button.


---
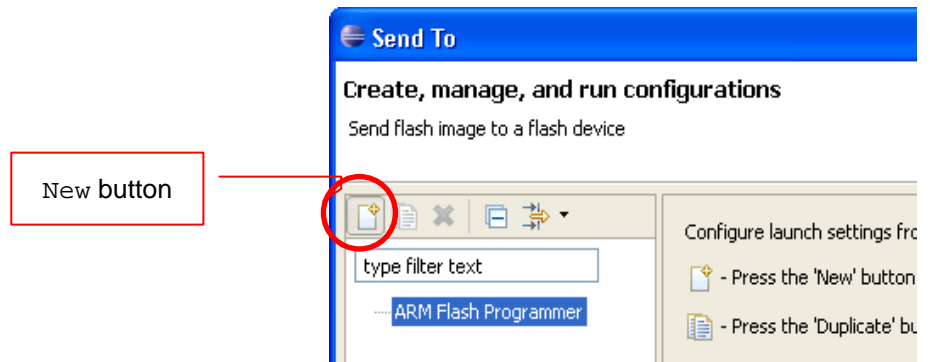
3   Ensure that you are in the `Image` tab. Enter a name for the new configuration in the `Name` field. Confirm that the `Select project` field is populated with the name of your project and that the `Select image` field is populated with the location of the U-Boot image within your project. Set `Select file type` to `Binary Image` to force the flash programmer to write the image at a set location in flash. Set `Start address` to the beginning of the first free block in flash (0x34040000 in the example in **Section 3.2.2 Step 4**). Press Apply.
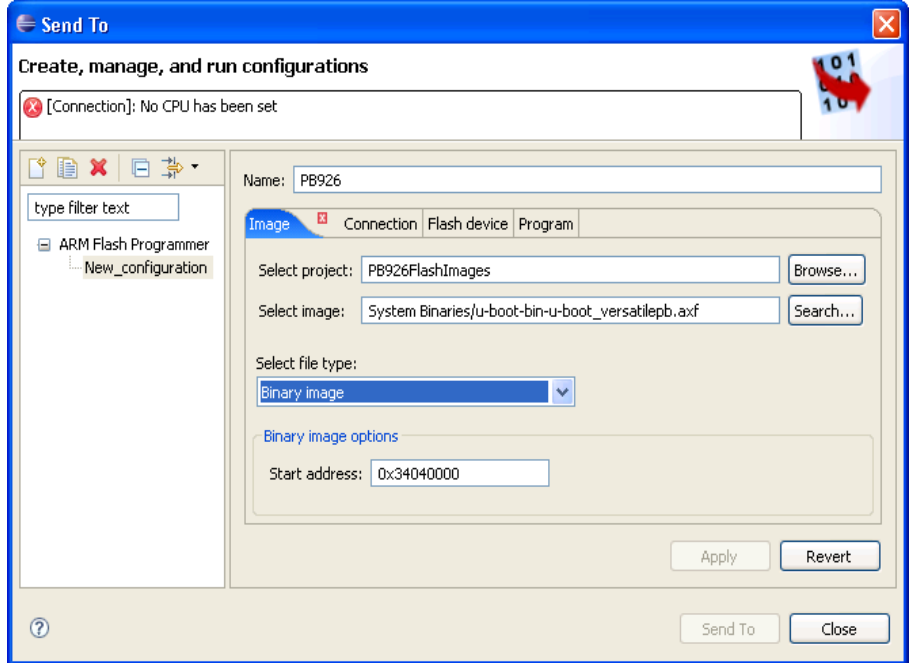
4   Select the `Connection` tab and press the `Configure...` button. Configure the connection to your hardware target as normal (see **Working with the ARM Flash Programmer** in the **RealView Development Suite Eclipse Plug-ins User Guide**).

5   Select the `Flash device` tab and select your target type in the `Flash device target configuration` drop-down for example `ARM Versatile PB926EJ-S`. Select the flash device to use, for example all of the `NOR` devices.

                   Application Note 201
                                                                                              ARM DAI 0201A
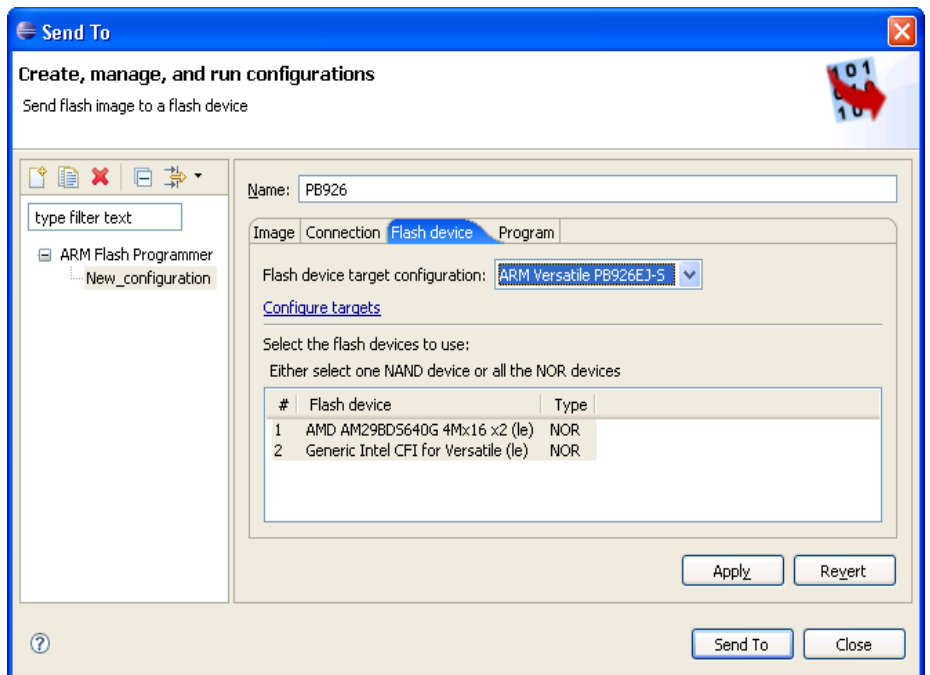
6 Open the `Program` tab. Ensure that `Erase only modified blocks` and `Verify after programming` are selected.
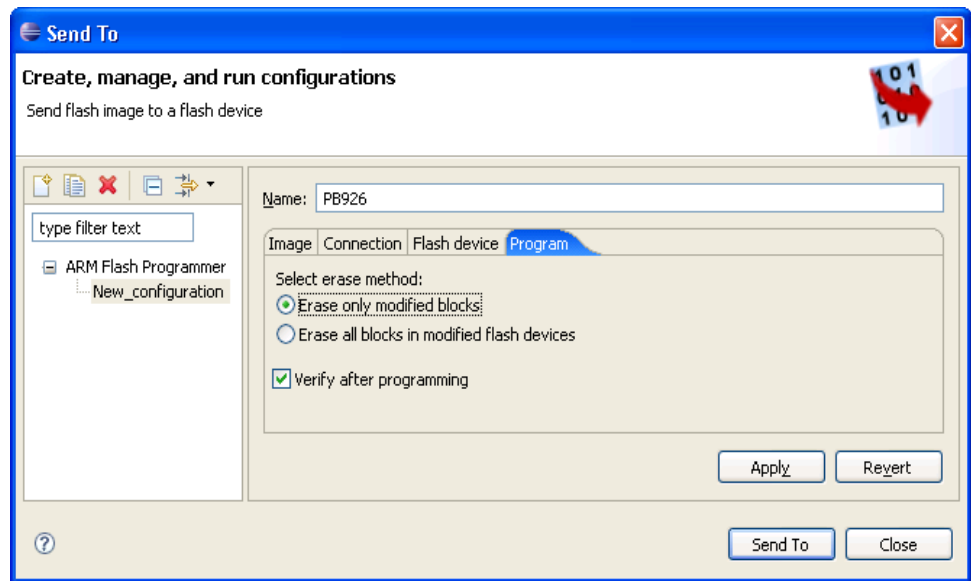
7 Press the `Send To` button. The flash programmer will flash the U-Boot image to the target. When it completes it will tell you how many blocks it has written. Use this information to compute the next free block. In the example above, 0x34040000 is the beginning of an area of flash with a block size of 0x40000. Therefore, if the U-Boot image requires 2 blocks then the next free block starts at address 0x340C0000 (0x34040000 + 0x80000).

Program Kernel Image

Re-open the `Send To` dialog and select the configuration that you have just used to flash U-Boot to the target. Change the `Select image` field to your Linux kernel image (for example `System Binaries/kernel-bin-2.6.24-arm2-uImage-2.6.24-arm2-versatile` if you have downloaded the image, or `linux-2.6.24/arch/arm/boot/uImage` if you have built it yourself). Change the `Start address` field to the address of the next free block in flash. Press `Send To` to flash the kernel image to the target.
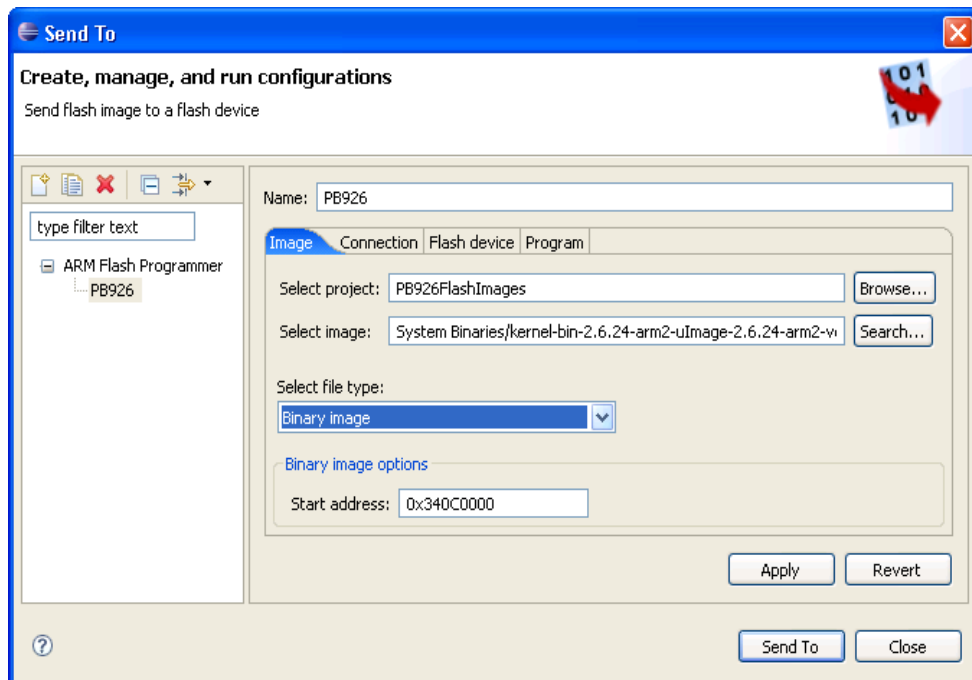
Program File System Image

Repeat the above process to flash one of the file system images (for example, `armbase_2.5.cramfs`) to the next free block in flash after the kernel image.
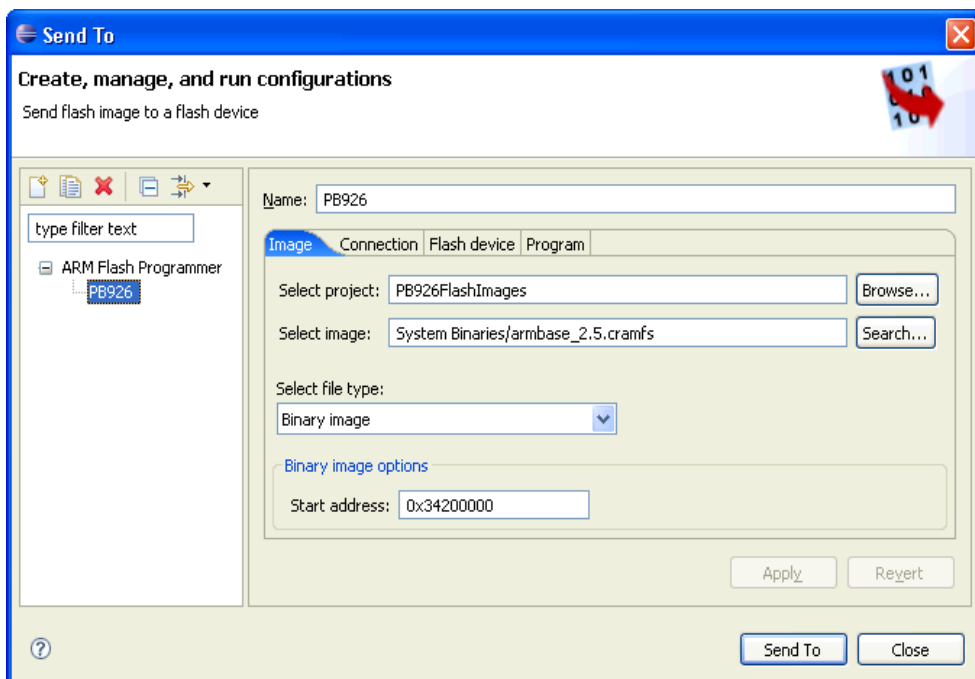
Note Boot Monitor will be unaware of any images programmed to flash using the ARM Flash Programmer. Using BootMonitor to program flash will cause these images to be overwritten. Therefore, having used the ARM Eclipse Flash Programmer, you should not use Boot Monitor for any further flash programming unless you take care to force Boot Monitor to write to regions of flash that you know to be free.

Note The flash configuration on your board may differ from that expected by the ARM Flash Programmer if you are using a recent revision of your board. This is more likely to be a problem is you are using flash devices with a higher address in memory than the first flash device on the target. If you need to modify the ARM Flash Programmer's flash target configuration then you can do so by following the `Configure targets` link on the `Flash device` tab in the `Send To` dialog and creating and modifying a copy of the built-in definition of your target. Press `F1` while in this dialog to access the online help that will guide you through this process.

3.2.4   Configure U-Boot

Note Some of the commands given in this section are word-wrapped. Each command should be entered as a single line. Points in commands where you should enter variable data are indicated by angle brackets around descriptive text (`<angle brackets>`).

1   Reset your board.

2   At the Boot Monitor prompt type:

```
> debug go <location of U-Boot>
```

In the example above, `<location of U-Boot>` would be 0x34040000.

3   U-Boot will begin to execute. Press any key to stop it from auto-booting.

4   Configure the arguments that U-Boot will pass to the kernel. U-Boot passes the contents of its bootargs environment variable to the kernel command line. To set this environment variable type the following:

```
setenv bootargs root=/dev/mtdblock0
mtdparts=armflash.0:<cramfs size>@<cramfs offset>(cramfs)
ip=dhcp mem=128M console=ttyAMA0
```

This command will tell the kernel to use serial port 0 (`ttyAMA0`) as the system console, to use 128MB of system RAM for the kernel and to use DHCP to get an IP address. `root=/dev/mtdblock0` tells it to read its root file system from flash, and the `mtdparts` command tells it where in flash the root file system is.

`<cramfs offset>` should be set to the offset from the start of flash where the root file system begins, and `<cramfs size>` should be set to the total size of the root file system. If you have used the `armbase_2.5.cramfs` file system then `<cramfs size>` will be 7442432 bytes. If the file system begins at address 0x34200000 in flash, and flash begins at 0x34000000 then <cramfs offset> will be 0x200000 (0x34200000 - 0x34000000). You should therefore enter the following command:

```
setenv bootargs root=/dev/mtdblock0
mtdparts=armflash.0:7442432@0x200000(cramfs) ip=dhcp
mem=128M console=ttyAMA0
```

5   Set the `bootcmd` environment variable to tell U-Boot how to boot the kernel. The necessary command is:

```
setenv bootcmd cp.b <kernel address> 0x7fc0 <kernel size>\;
bootm
```

where `<kernel address>` is the location of the kernel U-Boot image in flash memory and `<kernel size>` is the size of the kernel U-Boot image. In the

above example, `<kernel address>` is 0x34100000 and `<kernel size>` is 0x13119c.

6   Type `saveenv` to save the U-Boot environment variables in flash.

Note   The offset following the `@` symbol in the `mtdparts` parameter of `bootargs` is the offset from the beginning of flash memory, not the offset from the beginning of the area of flash within which the file system resides.

Note   The `(cramfs)` part of the `mtdparts` parameter of `bootargs` is simply a label for the MTD device. You may place any label you like between the brackets (for example `(root)` or `(filesystem)` might also be appropriate).

Note   If mounting a writeable file system (for example JFFS2), the MTD device must occupy an area of flash with a constant block size for you to be able to write to it.

3.2.5   Boot the System

Note   The commands given in this section are **not** word-wrapped. Line-breaks in the command indicate that you should press return. Points in commands where you should enter variable data are indicated by angle brackets around descriptive text (`<angle brackets>`).

After U-Boot has finished saving its settings and returns to the prompt, press the reset button again. Run U-Boot from the Boot Monitor prompt again as above:

```
debug go <location of U-Boot>
```

Your Linux system should now boot. You should be able to interact with the system either by using a VGA monitor, keyboard and mouse or through the serial console. Log in as `root` with no password.

To automate system startup in the future, it is possible to set up a boot script within Boot Monitor to invoke U-Boot automatically at reset. Please refer to your development board user guide for instructions and DIP switch settings. You should be able to find your development board user guide on the **Versatile Family CD**, or at http://infocenter.arm.com under **Development Boards->RealView Versatile baseboards**. In the case of the PB926, follow these steps:

1   Reset the PB926.

2   If you have a lead-free board then you should install the boot script in NOR flash. If your board is not lead-free then you should install the boot script in DoC flash.

To install the boot script in DoC flash, type the following at the Boot Monitor prompt:

```
> create <script name>

debug go <location of U-Boot>
```

To install the boot script in NOR flash, type the following at the Boot Monitor prompt:

```
> flash create <script name> flash_address <address>

debug go <location of U-Boot>
```

In both cases, `<script name>` is the name for your boot script (for example `boot.txt`). Note that there is a newline between the command that tells Boot Monitor to create a script and the script content. You should also follow the `debug go` line with a newline.

If you are programming into NOR flash then `<address>` must be a location in flash where you have not already written your system files (for example, somewhere after the end of your root file system). In the example that we have

been following in this chapter, 0x34940000 would be a suitable value for `<address>`.

3   Press `Ctrl-Z` to indicate the end of the boot script and return to Boot Monitor.

4   Verify that the file was entered correctly by typing:

```
> type <script name>
```

5   Specify the boot script to use at reset from the Boot Monitor by typing:

```
> set bootscript <script name>
```

6   Set S1-1 ON to instruct the Boot Monitor to run the boot script at power on.

7   Reset the PB926. The Boot Monitor runs and executes the boot script. In this case, it runs the U-Boot image which will, in turn, boot your Linux kernel.

4 Kernel Debug with RealView Debugger

Debug of the Linux kernel can be accomplished using RealView Debugger in conjunction with RealView ICE. RVD can only debug the Linux kernel itself: this includes kernel modules that are statically built into the kernel, but not loadable kernel modules.

Loadable modules and Linux applications cannot be debugged using RVD because modules and applications are reliant on a relocation process that only takes place when they are loaded by the kernel. Furthermore, RVD cannot distinguish between different Linux application processes as they all execute code from the same virtual address. For more details see the FAQ **Can I debug Linux Applications and Kernel modules using RVD?** at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/14250.html. We recommend that all modules that you wish to debug are built into the kernel statically and that GDB is used for application debug (see **Section 5**).

4.1 Requirements

This section assumes that you have built a Linux kernel image, or that you otherwise have access to both a debug-enabled kernel binary and an associated image file containing debug symbols.

If you have built the kernel image yourself then it is important that debug information is turned on. This can be activated by selecting the `Compile the kernel with debug info` option in the `Kernel hacking` submenu of the kernel configuration. This is the default in the kernel configurations available from http://www.arm.com/products/os/linux_download.html and in the default configuration created when the ARM patch is applied to the Linux source tree.

As noted above, all parts of the kernel that you wish to be able to debug must be compiled into the kernel itself. RVD cannot debug loadable kernel modules.

## 4.2 Debugging a Running Kernel Using RealView Debugger

This section assumes that you are attaching an RVD session to a running kernel. Using RVD to boot a kernel on a target board is covered in **Section 4.3**.

4.2.1 Set up the Debug Connection

1   Start RVD.

2   Create a debug connection to the target hardware as normal
    (`Target -> Connect to target...`). See the RealView Debugger User Guide for more information about creating debug connections in RVD.

3   Open up the debug connection, right-click on the core that you will attach the debugger to and select `Properties` from the context menu to open the `Connection Properties` dialog.

4   Open up the selected connection in the tree view (left-hand pane) of the `Connection Properties` dialog.
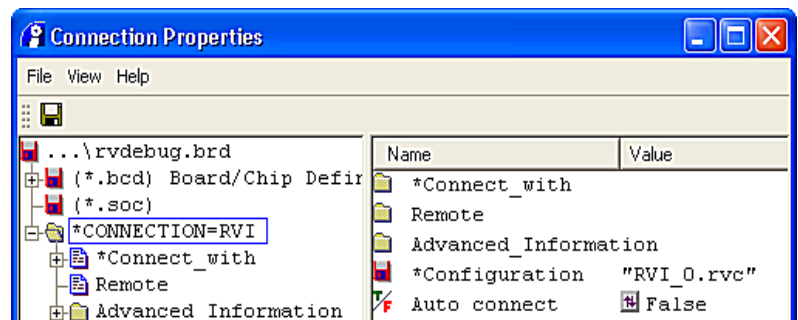
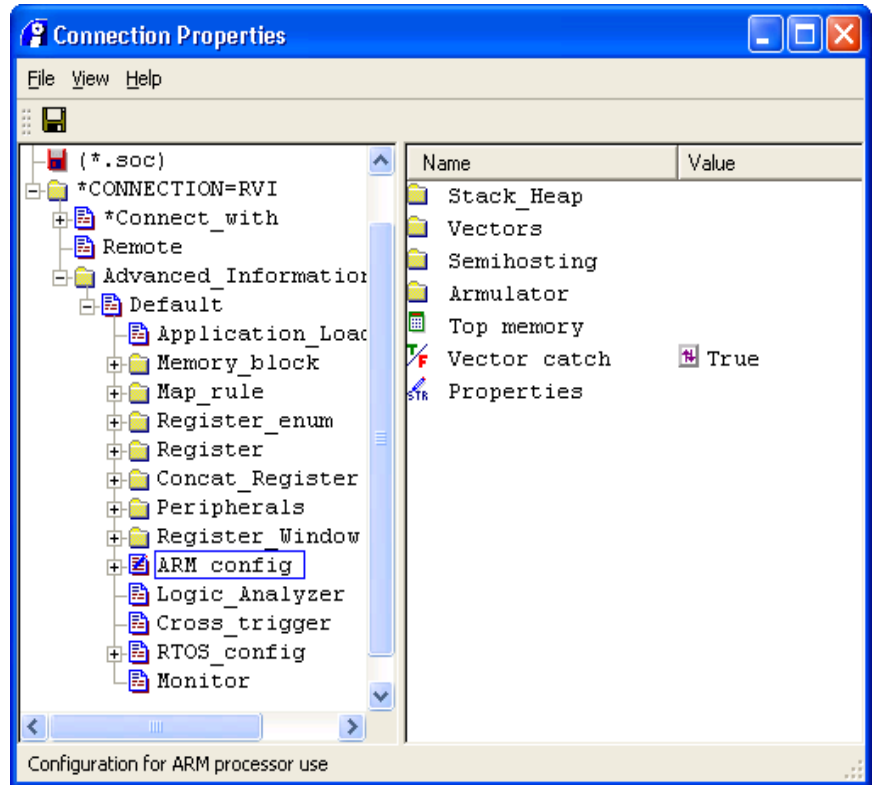5   Still in the tree view, open up `Advanced Information` and select `Default`.
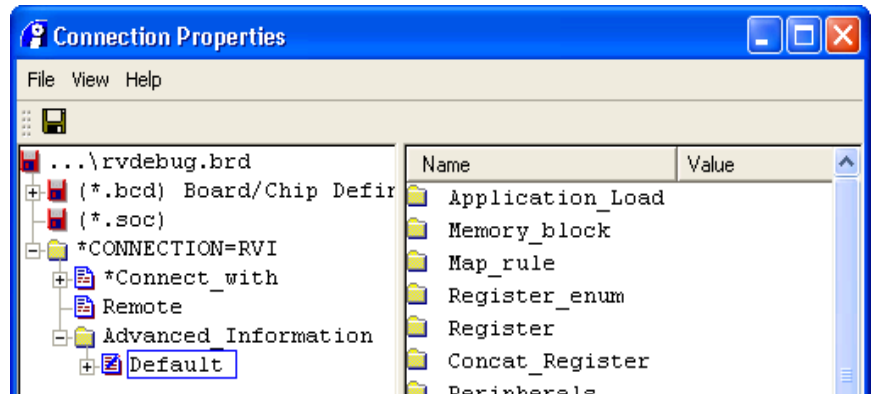
6   In the right-hand pane, right-click on `Connect mode` and select
     `no_reset_and_no_stop`. This will prevent RVD from resetting or stopping the
     target when it connects to it.

       Application Note 201
                                                                                                ARM DAI 0201A
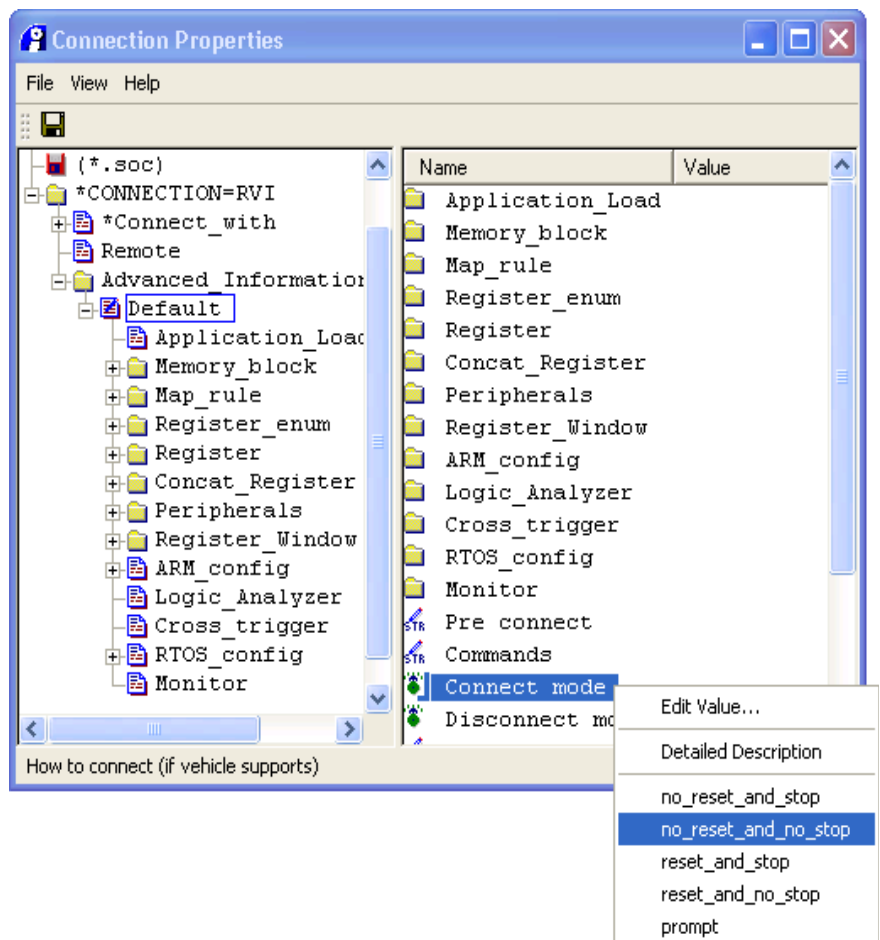
7   In the tree view, open up `Default` and select `ARM_config`.

8   In the right-hand pane, set `Vector catch` to `False`. This will prevent RVD from intercepting exceptions. Exceptions can occur in the normal operation of the kernel and will be handled by it.
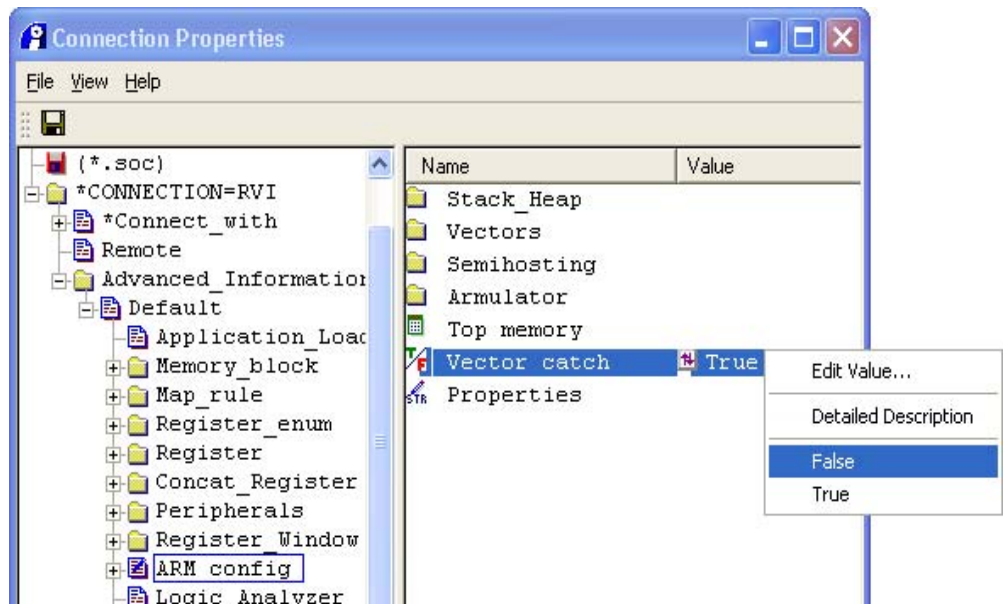
9 In the tree view, open up `ARM_config` and select `Semihosting`.
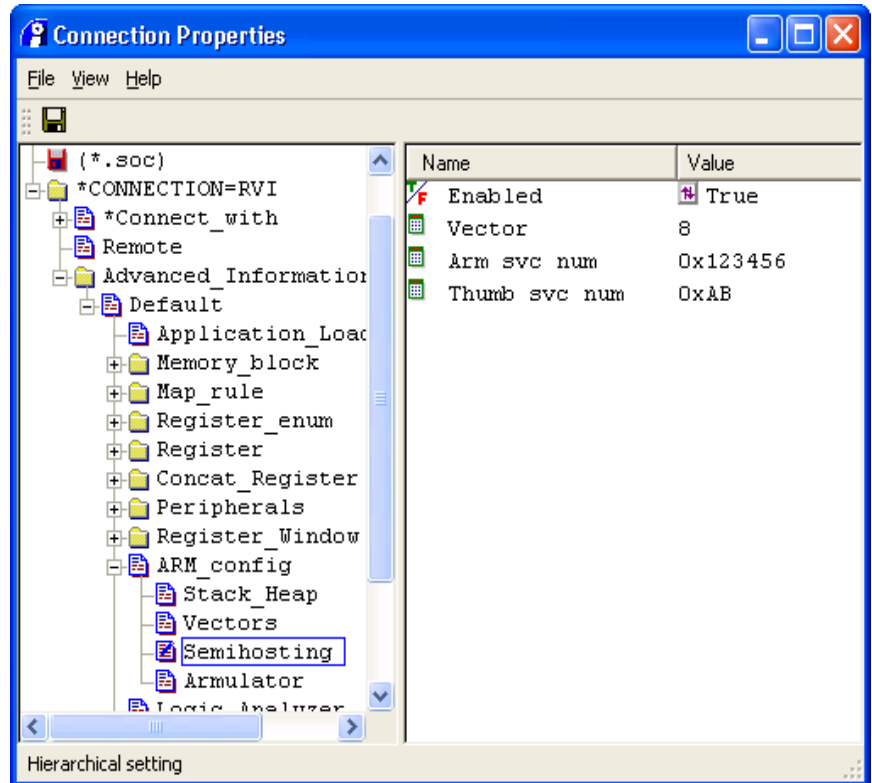
10 In the right-hand pane, set `Enabled` to `False`. The Linux C runtime does not use semihosting and so RVD should not be attempting to provide semihosting services.

11 Save the changed connection properties and close the dialog.

12 The debug connection can now be used to connect to the target.

Note If you have set a Board/Chip definition file for this connection, then this file might contain values for this connection that override the target connection settings that you have made in this section.
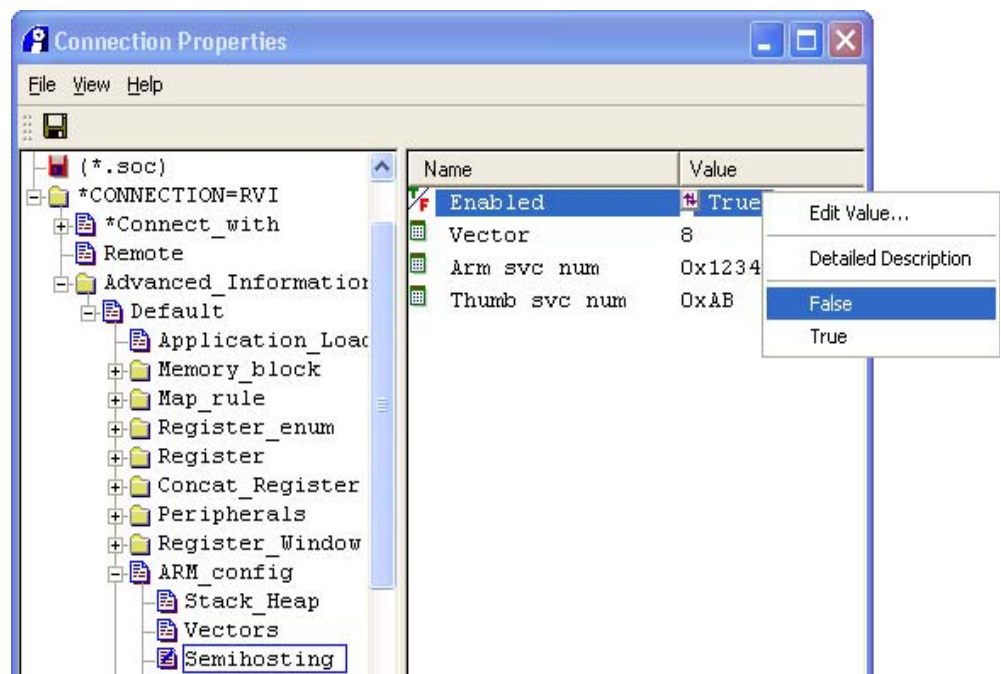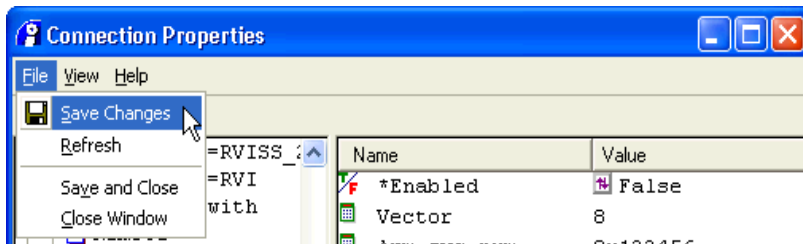
4.2.2 Load Debug Symbols

1 If you have not done so already, connect to the target.

2 Open the `Load Image` dialog by selecting `Load Image...` from the `Target` menu.

In the `Files of Type` drop-down, select `All files (*)`.

Tick the `Symbols Only` check box. It is the responsibility of your boot loader to load the kernel code: you do not want RVD to load it again.

Ensure that `Auto-Set PC` and `Set PC to Entry Point` are not ticked. Your system is already running, so you do not want RVD to change the PC.

Navigate to the location of your kernel source tree and select the `vmlinux` file that is at its root. This file contains the kernel's debug information.
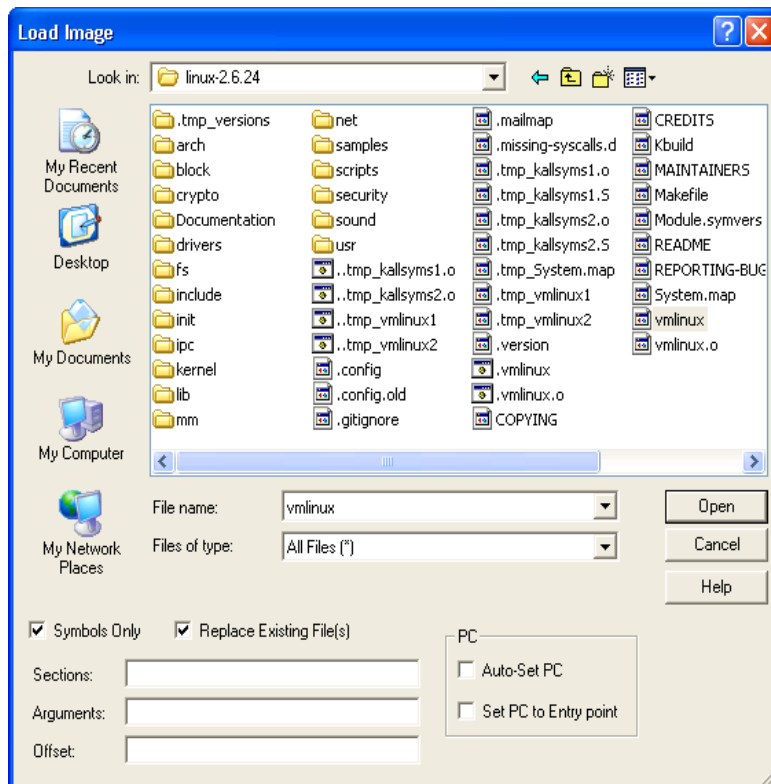
3 Press the `Open` button. RVD will load the debug symbols for the Linux kernel. This may take some time.

 Application Note 201
ARM DAI 0201A

    4   RVD now has the kernel debug information loaded and is able to proceed with debugging in the usual manner.

4.2.3   Debug the Kernel

You can debug the kernel as you would any other software running on the target. However, there are some issues to be aware of when using RealView Debugger to debug the Linux kernel.

1   Only statically compiled kernel code can be debugged with RVD, not Linux applications and not loadable kernel modules (see the FAQ **Can I debug Linux applications and Kernel modules using RVD?** at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/14250.html).

2   Software breakpoints can be set at any point after the MMU has been initialized, which means at any point from `MAIN\start_kernel` onwards. **The RealView Debugger User Guide** has full information about setting breakpoints with RVD, but as an example, typing the following in the `Cmd` window will set a breakpoint at the beginning of `MAIN\start_kernel`:

```
binstr &&@vmlinux\\MAIN\start_kernel
```

There are no restrictions on when hardware breakpoints (set with the `bexec` instruction) may be set.

3   RVD does not provide any special awareness of the Linux kernel and will treat it as it would any other software running directly on the hardware target. It is therefore not possible to tell, for example, which process was running when a kernel breakpoint is triggered, or even if two consecutive breakpoint hits were caused by the same process.

4   Unless you are able to configure your kernel to build with no optimizations and full debug information, you will not get a perfect debug illusion. For example, the debugger may appear to skip source lines, or to be executing a different source line from the one that is actually currently being executed. Note that the ARM Embedded Linux kernel build does not permit all optimizations to be disabled. Although the `General setup` submenu of the kernel configuration has an `Optimize for size option`, disabling this option causes the build to optimize for performance instead of code size, so the kernel will still contain optimized code

4.3   Booting the Kernel with RealView Debugger

When developing and debugging the kernel, it is useful to be able to boot the kernel using RealView Debugger rather than having to install each new kernel and load it with a bootloader such as U-Boot, as described in **Section 3**. **Figure 4-1** shows an RVD script that will boot a kernel on a PB926 board with a debug connection set up as described above. It assumes that the root file system is a cramfs file system stored in flash at offset 0x200000 and with size 7442432 bytes. To use this method you must first have run the appropriate initialization script for your board. In the case of the PB926 this script is `$ARMROOT/Versatile/Firmware/3.4/2/Misc/VPB926EJS_SDRAM_Init_rvd.li`. Suitable scripts for other boards can be found in the same location.

Figure 4-1: RVD Linux Kernel Boot Script

```
//load kernel binary image
readfile,raw "H:\linux-2.6.24\arch\arm\boot\Image"=0x8000

//load kernel symbols
load /ni "H:\\linux-2.6.24\\vmlinux"

@r0=0 //must be 0
@r1=387 //machine ID of target
@r2=0x100 //start of ATAG list
@pc=0x8000 //set pc to first instruction in kernel

//clear the first page
fill 0x0000..0x0fff = 0

//ATAG_CORE header
setmem /32 0x100 =2 //size of ATAG_CORE (in words)
setmem /32 0x104 =54410001h //id of ATAG_CORE
//no ATAG_CORE content required

//ATAG_CMDLINE header
setmem /32 0x108 =258 //size of ATAG_CMDLINE (in words)
setmem /32 0x10c =54410009h //id of ATAG_CMDLINE

//ATAG_CMDLINE content
setmem /8 0x110 ="root=/dev/mtdblock0
mtdparts=armflash.0:7442432@0x200000(cramfs) ip=dhcp mem=128M
console=ttyAMA0"

go
```

The first line of **Figure 4-1** loads the raw, uncompressed kernel image at address 0x8000.

The second line loads the debug symbols for the image. The `/ni` option selects load of symbols only - the `readfile` command on the first line has already loaded the code. Note that it is necessary to escape the Windows path character for the `load` command, but not for the `readfile` command.

The next lines set registers to the values that they must have when control passes to the kernel. The main ones to note are `r1`, which must be set to the machine ID of the target, and `r2`, which must be set to the start of the ATAG list in memory. The machine ID 387 identifies the RealView Versatile Platform Baseboard (for example the PB926). Machine ID 827 identifies the RealView Emulation Baseboard. All Linux machine IDs can be found at http://www.arm.linux.org.uk/developer/machines.

Finally, we fill the first page of memory with zeros, then use memory starting at 0x100 to initialize the kernel parameters. These consist of a list of structures called ATAGs. We use these to set the kernel's command line. The ATAG_CORE is required to start the list.

The command line string provided for the ATAG_CMDLINE itself is just the same as the command line described in **Section 3**, above. It must be null-terminated, but as we have already initialized the first page of memory to 0 then we can assume that it will end with a zero byte. Similarly, we can set a large size of ATAG_CMDLINE so long as it is the last ATAG that we set. The only restriction is that we must leave at least two zero-value bytes to serve as an ATAG_NONE.

The ATAG list must be terminated by an ATAG_NONE. However, as an ATAG_NONE consists of two words with zero value, this has effectively been provided by filling the first page with zeros.

To adapt this script for your own use, all that is required is to fix the paths for the kernel image and debug symbols to match your own system, to set $r2$ to the appropriate machine ID for your target and to modify the command line given in ATAG_CMDLINE to match your target. You will not have to modify the size of ATAG_CMDLINE unless you exceed the existing high limit.

5 Application Debug with Eclipse & GDB

Debug of applications running on an ARM Linux target can be accomplished using Eclipse in combination with the CodeSourcery GDB and gdbserver applications.

5.1 Debugging with GDB and gdbserver

A GDB client on the host system can be using in conjunction with a gdbserver process running on the target system to achieve application debug on ARM Linux systems. The host-side GDB configuration can be handled using Eclipse, but the gdbserver process must be started manually on the target. The same application image must be available on both the host and the target. The copy of the image on the host must not be stripped, but the copy of the image on the target may be stripped. The images must not otherwise differ: in other words, the image on the target must be either an exact copy of the image on the host, or else a stripped copy of the image on the host.

Figure 5-1 depicts the communication between the main software components in a typical Eclipse/GDB/gdbserver debug session.

Figure 5-1: Debugging an ARM Linux Application
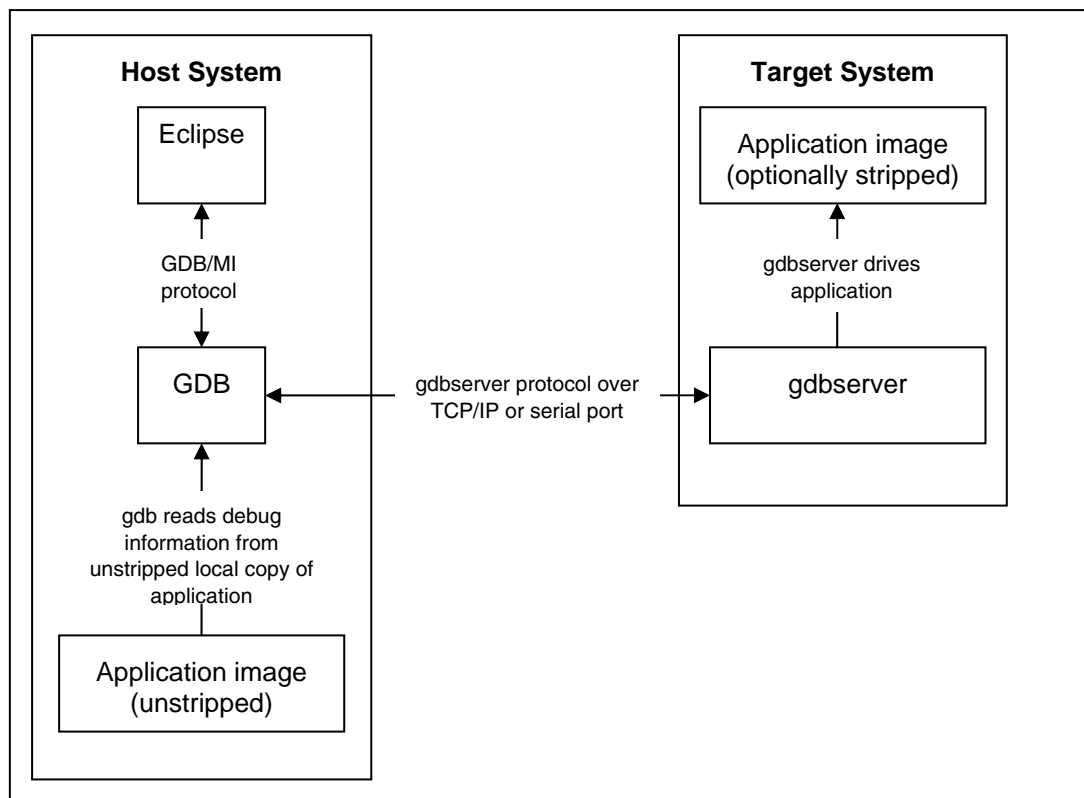
5.2 Requirements

5.2.1 Target Requirements

This section assumes that you have an ARM Linux target with the following:

1. A gdbserver binary

2. A copy of the application to debug, including any shared libraries on which it depends

---

A build of gdbserver can be found in all versions of the ARM Embedded Linux file system, but it is recommended that you use the gdbserver that was built along with the rest of your GDB installation if this is possible. In the case of the CodeSourcery 2007q1-21 toolchain, gdbserver can be found at `<CodeSourcery Root>/arm-none-linux-gnueabi/libc/usr/bin/gdbserver`. See **Section 5.3.1** for information about ways that files can be transferred from the host system to the target system.

The application and shared libraries on the target side may be stripped, but it is harmless to leave them unstripped.

5.2.2 Host Requirements

This section assumes that you have the following on your host:

1. The Eclipse supplied with RealView Development Suite 3.1
2. A GDB that consumes ARM binaries
3. Copies of the application and shared libraries being debugged on the target

The Eclipse supplied with RealView Development Suite 3.1 is composed of the Eclipse Platform version 3.2, the Eclipse C/C++ Development Tools version 3.1.1 and the RealView Development Suite plug-ins. The RealView Development Suite plug-ins are not required to follow the instructions in this section.

GDB is assumed to be the GDB binary supplied with the ARM Linux version of the CodeSourcery toolchain.

The application and any shared libraries that you wish to debug into must have their debug symbols available, so you must have a build with debug information to keep on the host side. If you want source-level debug then you must also have the source available on the host system.

Note   It is important that copies of applications and shared libraries on the host are identical to the copies on the target: they *must* be from the same build. The only permissible difference is that the copies on the target may have their debug symbols removed, for example by using the `strip` tool. Failure to keep the copies synchronized will result in unpredictable debug behavior.

5.3   Preparing the Target

5.3.1   Transfer Files to the Target System

It is necessary to transfer the images for debug, and possibly a gdbserver binary, onto the target file system. If you are using any kind of networked file system then this may be as simple as copying the files into a location on the host side that can be mounted on the target side. If this is not possible then you will need to be able to transfer the files to the target system. All versions of ARM Embedded Linux provide wget and an FTP client. Therefore, if you are running an FTP server on your host then you will be able to run the FTP client on your target to get files from your host, or alternatively you can download files available via HTTP by using wget.

5.3.2   Run gdbserver

You must be running a gdbserver process on the target system. You will connect to gdbserver from the host system in order to debug your application. You may start the target application using gdbserver, or you can attach gdbserver to an already-running process.

The syntax for launching an application with gdbserver is:

```
gdbserver <comm> <prog> <args>
```

where `<comm>` tells gdbserver where to listen for connections, `<prog>` is the path to the binary to be executed and `<args>` are the command-line arguments to pass to the binary. `<comm>` may be the name of a serial port (for example `/dev/ttyS0`) or a port number preceded by a colon (for example `:5000`). gdbserver will bind to all available network interfaces.

A concrete example is the case where you have a `less` binary in your home directory and wish to start it with the `-M` option and listen for debug connections on port 5000 of the target. To do this, type:

```
gdbserver :5000 ~/less -M
```

GDB can also attach to a process that is running on the target. The syntax for attaching gdbserver to a running process is:

```
gdbserver <comm> --attach <pid>
```

where `<comm>` tells gdbserver where to listen for connections, as before, and `<pid>` is the process ID of the process to be debugged. Therefore, if we have a running `less -M` with process ID 100 and we want, again, to listen for connections on port 5000 of the target the command to type would be:

```
gdbserver :5000 --attach 100
```

Note   The process ID of a running process can be obtained using tools such as `pidof`, `top` and `ps`. See the man pages for these tools for details of how to use them.

5.4   Preparing the Host Debugger

5.4.1   Create a GDB Command File

Due to limitations in the Eclipse debug configurations it is necessary to create a GDB command file before beginning. This is just a text file that sets some options within GDB and you can give it any name and location on your host system that seems sensible. You will tell your Eclipse debug connection where to find the command file in **Step 6** of **Section 5.4.2**. When Eclipse launches GDB, it will instruct it to read the commands in this file. The GDB command file should read as follows:

```
set remotetimeout 0
set solib-absolute-prefix <CodeSourcery Root>/arm-none-linux-
gnueabi/libc/
```

where `<CodeSourcery Root>` is the root of your CodeSourcery installation. Note that the slash-direction is important here: GDB expects the directory separator to be a forward slash (`/`), even when running on Windows.

The first command in the file tells GDB not to time out while waiting for responses from gdbserver. The second command tells it where on the host to find debug builds of the shared libraries that are on the target system. If you are not using CodeSourcery's GNU libraries then provide the path to your host's copy of the target's system libraries. If you wish to be able to debug into the system libraries then the host's copies of the libraries must be built with debug information. Also, if you wish to do source-level debug of the libraries then you must have their source code on your host.

Note   Note that the Eclipse Debug Connection interface appears to provide a GUI-based means of setting the location of shared libraries. However, in the version of the Eclipse CDT released with RealView Development Suite 3.1, this part of the interface does not function correctly. This is a bug in the CDT over which ARM has no control. Future releases of RealView Development Suite should include a version of the CDT in which this bug is fixed.

5.4.2 Create an Eclipse Debug Connection

To create a debug connection in Eclipse you must have an Eclipse project to associate it with. If the application that you are debugging was created using an Eclipse project then you can use this as the project to associate the debug connection with. Otherwise, you can create an empty Eclipse C/C++ project to associate the debug connection with. To do this, select `New->Standard Make C++ Project` from the `File` menu to bring up the `New Project` dialog. Give the project a name and click `Finish`.

Having ensured that you have a project to associate your debug connection with, proceed as follows:

1   Ensure that you are in the `C/C++ Perspective`. If you are not, you can change to it by selecting `Open Perspective` from the `Window` menu.


Shows `C/C++ Perspective`

2   Right-click on your project in the `C/C++ Projects` view and select `Debug As->Debug...` from the context menu to open the `Debug` dialog.

3   Select `C/C++ Local Application` and press the `New` button at the top-left of the panel.

`New` button
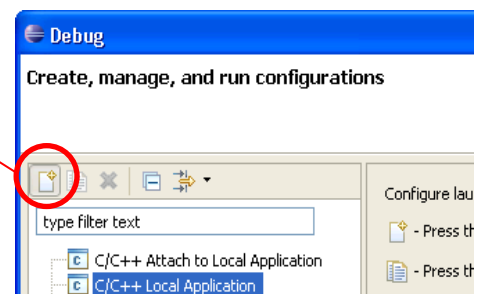
4   Ensure that you are in the `Main` tab. Enter a name for the new debug connection in the `Name` field. If your project contains the executable to be debugged then press the `Search Project...` button and select the image to be debugged. Otherwise, press the `Browse...` button and locate the image in your host's file system. In either case, the image selected must be an unstripped version of the image to which gdbserver is attached.
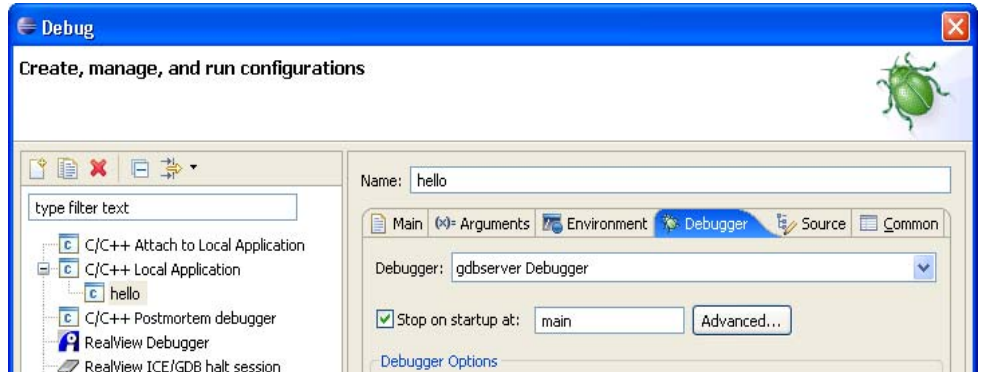

---
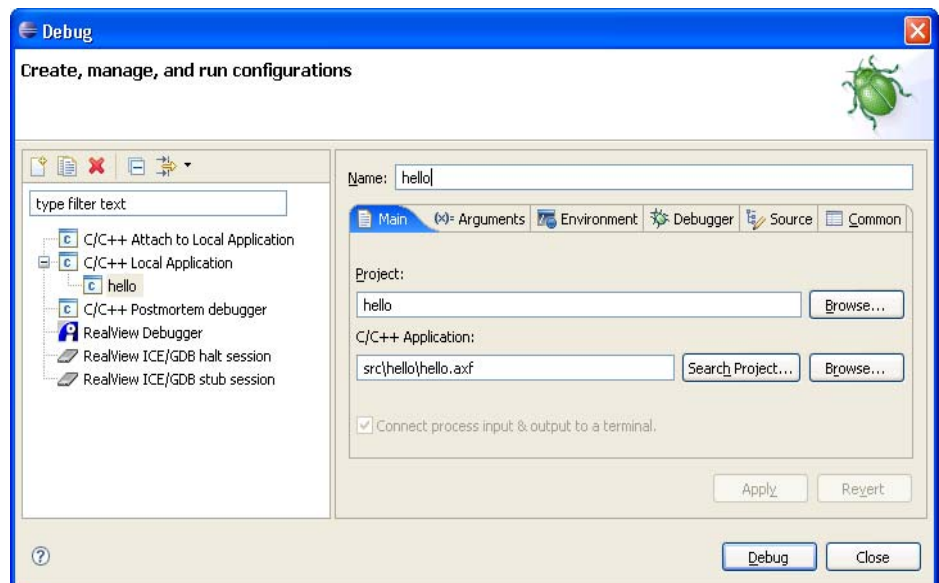
5   Go to the `Debugger` tab.

In the `Debugger` drop-down select `gdbserver Debugger.`

If your gdbserver session is launching a process (rather than attached to an existing process) then you may wish to select the `Stop on startup at` checkbox to stop the executable when it enters its `main` function.

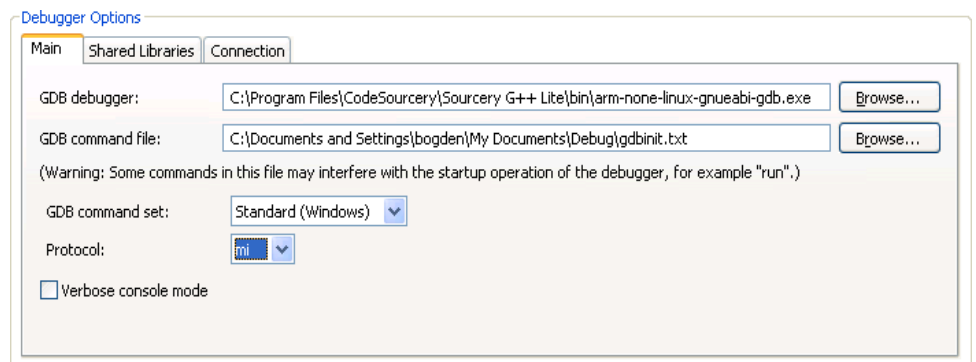6   In the `Debugger Options` pane, select the `Main` tab.

To fill the `GDB debugger` field, press the `Browse...` button and navigate to the location of your GDB debugger. If you are using CodeSourcery's GDB build then this will be at
`<CodeSourcery Root>/bin/arm-none-linux-gnueabi-gdb`, where `<CodeSourcery Root>` is the root of your CodeSourcery installation.

To fill the `GDB command file` field, press the `Browse...` button and navigate to the location of the GDB command file that you created in **Section 5.4.1**.

Ensure that `GDB command set` is set to `Standard` if your host system runs Linux or to `Standard (Windows)` if your host system runs Windows.

Ensure that `Protocol` is set to `mi`. This refers to the GDM/MI interface that Eclipse uses to drive GDB on the host machine. Using the original version of the interface (just plain `mi` in the `Protocol` drop-down) ensures that all versions of GDB dating from the introduction of GDB/MI can be driven by your debug configuration.

        Copyright © 2008 ARM Limited. All rights reserved.

7   In the `Debugger Options` pane, select the `Shared Libraries` tab and ensure that `Load shared library symbols automatically` is ticked.

Do not add shared directories to the list of shared libraries: as noted above, this does not work in the version of the CDT included in the Eclipse plug-ins shipped with RealView Development Suite 3.1. The locations of shared libraries must, instead, be specified in the GDB command file.

8   In the `Debugger Options` pane, select the `Connection` tab and fill in the information required to connect to gdbserver on your target system, as follows:

If you are connecting over a serial port then:

Set `Type` to `Serial`

Set `Device` to the serial port on the host system (for example `/dev/ttyS0` or `COM1`)

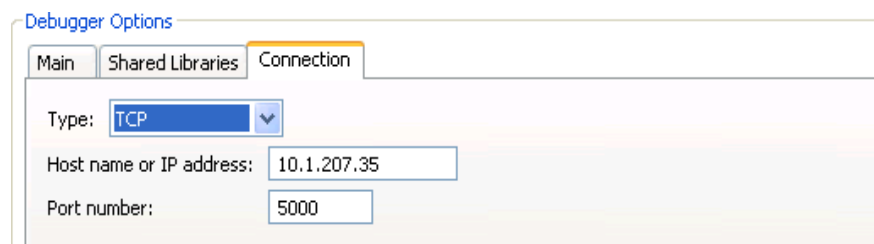Set `Speed` to the speed of the serial port

If you are connecting over TCP then:

Set `Type` to `TCP`

Set `Host name or IP address` to the host name or IP address of the target system

Set `Port number` to the port on which the gdbserver process on the target is listening

9   Press the `Debug` button to connect to the gdbserver session and begin debugging, or the `Apply` and `Close` buttons to save the changes and return to Eclipse.

There are several other tabs available to provide information about the Eclipse Debug Connection, but most of the available connection information is irrelevant in the gdbserver case. For example, the `Arguments` and `Environment` tabs are irrelevant because the application has already been launched before Eclipse connects to it.

5.5   Debugging the Application

You can now debug as normal. Note that shared library debug symbols may not be loaded until the shared library itself is loaded: therefore, it may not be possible to set breakpoints in shared libraries until after they are loaded. If you want to be able to do this then ensure that you have `Stop on shared library events` ticked in the `Shared Libraries` tab of the `Debugger Options` pane in the `Debug` dialog. This will suspend the debug target when shared libraries are loaded, allowing you to set breakpoints.

It is important to be careful when rebuilding the target, especially if your Eclipse project builds the binary for you, and even more so if it builds the binary automatically when you launch your debug session. Eclipse will not transfer the new binary to the target system: you must do this yourself and restart any gdbserver session attached to it. If the binaries on host and target are not synchronized then debug behavior will be unpredictable.

Known Issues

1   Be aware that you will be able to see two output consoles in the `Console` view during debug sessions. One of these consoles will have a name ending with the name of application being debugged. This is for output from your application, but it will not work when debugging via gdbserver. Output will be displayed on the target system, not through Eclipse.

The other output console will have a name ending with the name of your GDB binary. This will display output from the GDB process and may be useful for diagnosing problems with GDB itself.

2   In some circumstances, GDB may not be properly terminated by Eclipse. To minimize the chance of this happening, always select the application being debugged in the `Debug` view before pressing the `Terminate` button. If your host's performance becomes very poor, check for hanging GDB processes.

3   When you terminate the debug session on the host, gdbserver will be terminated on the target. You will need to start a new gdbserver session on the target to be able to begin a new debug session.

6   References and Further Information

For further information on the GNU toolchain supplied by CodeSourcery, please see:

http://www.codesourcery.com/gnu_toolchains/arm/

In particular, the FAQ for the ARM GNU toolchain can be found at:

http://www.codesourcery.com/gnu_toolchains/arm/faq.html

ARM Embedded Linux is available from the ARM website at:

http://www.arm.com/linux/

To use RVCT to build applications for ARM Linux targets see **Application Note 178: Building Linux Applications Using RVDS 3.1 and the GNU Tools and Libraries**, available at
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0178a/index.html.

ARM support provides a number of FAQs about Linux:

- **How do I rebuild the Linux kernel for my ARM RealView development board?**: http://www.arm.com/support/faqdev/14409.html.

- **How do I install the pre-built Linux images on my ARM RealView development board?**: http://www.arm.com/support/faqdev/14586.html.

- **What technical support does ARM provide for Linux?**: http://www.arm.com/support/faqdev/18411.html.

- **Can I debug my Linux kernel with RVD?**: http://www.arm.com/support/faqdev/14249.html.

- **Can I debug Linux Applications and Kernel modules using RVD?**: http://www.arm.com/support/faqdev/14250.html.

- **How do I build Linux applications with RVCT 3.x?**: http://www.arm.com/support/faqdev/12750.html.

General information on ARM Linux can be found from the open-source community. Useful starting points are:

- The `comp.sys.arm` newsgroup

- The ARM Linux project website: http://www.arm.linux.org.uk/

- The ARM Linux wiki: http://www.linux-arm.org/

Application Note 201
ARM DAI 0201A